Süsteemprogrammeerimine keeles C

Loeng 9

Milles saame teada kuidas teha muutuva argumentide arvuga funktsioone, kuidas möllata protsessidega: zombidest ja deemonitest.

Eelmises osas (time)

Aja jaoks on andmetüübid:
time_t – hoiab sekundeid 01.01.1970 kesköösT
clock_t – hoiab protsessori *tick*e
struct tm – hoiab komponentideks löödud aega

clock() annab *tick*e, 1000000 sekundis
time() annab sekundid 1970 jaanuari algusest
gmtime(), localtime() teevad sekundeist UTC või kohalikku aega kujutava struktuuri tm

Eelmises osas (aeg (2))

Aeg stringiks:
ctime() - sekunditest
asctime() - struct tm alusel

mktime() - struct tm > sekundid; normaliseerib

Eelmises osas (*socket*)

Socket on koht, kuhu saab võrk kirjutada, lugeda.
Socketi loomisel saab määrata protokolliperekonna (nt PF_INET) ja pakettide iseloomu (SOCK_DGRAM, SOCK_STREAM, SOCK_RAW)

socket() - annab erilise failideskriptori, millel on read(), write(), send(), recv(), close()

Eelmises osas (connect)

Kui socket on olemas, tuleb öelda, kuhu ta ühendub: connect()

Keeruline osa on aadressi esitus: selleks eriline struktuur vastavalt protokollile, ning selle pikkus baitides.

Internetiühenduse puhul on aadress struktuuris sockaddr_in (üldkuju saab sockaddr nimelise kasutamisel)

Eelmises osas (baidijärjestus)

Kokkuleppeliselt on baitide järjestus võrgus nii, et kõige olulisem bait tuleb esimesena.

Port ja võrguaadress tuleb sellele kujule teisendada
htons() - host to network short
htonl() - host to network long
ntohs() - network to host short
ntohs() - network to host short

Eelmises osas (aadressid)

Hosti aadressi & info saamine:

gethostbyname() annab hostent struktuuri (sees ka aadress)

gethostbyaddr() annab hostent struktuuri olemasolevast etteantud pikkusega aadressistringist

Võrgundus(bind)

Bind määrab socketile kohaliku aadressi ja pordi mille juurde antud socket kuulub.

Ühenduste vastuvõtul on see oluline, ühendumisel on bind automaatne

Võrgundus (listen)

Selleks, et socketist ühendusi vastu võtta, tuleb neid oodata käsuga listen()

SOCK_STREAM ja SOCK_SEQPACKET puhul ainult

Argumendiks ka backlog, mis ütleb mitu teenindamata ühendust järjekorda mahutatakse

Võrgundus (accept)

accept() paneb programmi ühendust vastu võtma

kui ühendust ei ole (veel), blokeeritakse programmi töö kuniks tekib

datagramme saab niisamagi vastu võtta accept abil

accepti puhul luuakse uus socket

Võrgundus (send & recv)

send() saadab info socketisse, ütleb palju tegelikult saadeti.

recv() loeb socketist, ütleb palju tegelikult tuli

põhierinevus read() ja write() osas see, et on võimalik anda lisaparameetreid kuidas täpselt käituda

Eelmises osas (sendto ja recvfrom)

Datagrammidega on lihtsam

connect() ega accept() pole otseselt tarvilikud,
ühenduda saab ka sendto() ja recvfrom() abil.

Eelmises osas (ühenduse sulgemine)

close()

shutdown() - muudab socketi omadusi kas ainult vastuõtvaks, saatvaks, või keelates mõlemad. close() tuleb ikkaigi teha, et sulgeda.

Tänases osas

- Muutlik argumentide arv
- Protsessidega seotud süsteemikäsud
- Protsessi omanik ja grupp
- Protsessi identiteet (PID)

Varieeruva argumentide arvuga funktsioonid

- Varieeruva argumentide arvuga funktsioonide deklareerimiseks tuleb funktsiooni argumendid deklareerida nii, nagu me ei viitsiks neid kõiki välja kirjutada:

```
void foo( int arg1, int arg2, ...) {
```

- Kuidas neid argumente kätte saada?

Ülejäänud argumentide kättesaamine

Näide sehkendatakse kuskile teise kohta

Ülejäänud argumentide kättesaamine (lihtsam meetod)

- #include <stdarg.h>
- Failis on makrod/funktsioonid:

```
void va_start(va_list ap, last);
type va_arg(va_list ap, type);
void *va_end(va_list ap);
```

- Üks võimalik implementatsioon:

```
typedef char *va_list;
#define va_start(ap, v)  ((void) (ap = (va_list) &v + sizeof(v)))
#define va_arg(ap,type)   (*((type *)(ap))++)
#define va_end(ap)  ((void) (ap = 0))
```

va_start() initsialiseerib argumentide nimistu, last ei tohiks olla registrimuutuja, massiiv ega funktsioon.
va_arg() annab järgmise etteantud tüübist argumendi
va_end() teeb vajadusel puhastust

- Lisainfo: 'man stdarg' või 'man vararg'

Varieeruv argumentide arv (näide)

```
#include <stdarg.h>
#define MAXARGS     31
void f1(int n_ptrs, ...) {
    va_list ap;
    char *array[MAXARGS];
    int ptr_no = 0;

    if (n_ptrs > MAXARGS)
        n_ptrs = MAXARGS;
    va_start(ap, n_ptrs);
    while (ptr_no < n_ptrs)
         array[ptr_no++] = va_arg(ap, char*);
    va_end(ap);
    f2(n_ptrs, array);
}
```

Protsesside loomine ja hävitamine

- UNIX pakub nelja süsteemikäsku protsesside loomiseks, lõpetamiseks ja nende lõpetamise ootamiseks
  - exec() perekond
  - fork()
  - wait()
  - exit()

Protsessi mällu laadimine

- *Binary executable* koosneb harilikult päisest, (programmi)tekstist, andmetest, andmeteisaldusinfost ja sümboltabelist. Päist, sümboltabelit ja andmeteisaldusinfot kasutatakse korra ja siis visatakse minema, tekst ja info aga jääb mällu.

Executable file	Process memory
HEADER	TEXT
TEXT (program)	DATA
DATA (initialized)	(BSS)
BSS (=uninitialized data)	free mem
RELOCATION	STACK
SYMBOL TABLE (can be stripped)	USER BLOCK (in kernel adr space)

exec() käsuperekond

- Exec() käsud laevad käivitatava faili (*binary* executable) mällu protsessiks. Süntaks on järgmine:

```
extern char **environ;
int execl( const char *path, const char* arg, ...);
int execv (const char *path,  char *const argv[]);
int execle(const char *path,
          const char *arg, ..., char * const envp[]);
```

  - execl: täielik failinimi, var. argumendid charidena
  - execv: täielik failinimi, argumendid massiivis
  - execle: täielik failinimi, var. argumendid charidena, keskkond (*environment*)
- Mis on keskkond?

exec() perekond (2)

- Tegeliku töö teeb ära see funktsioon, mida teised kutsuvad:

```
int execve (const char *filename,
                char *const argv [],
                   char *const envp[]);
```

  - täielik failinimi, argumendimassiiv, keskkonnamassiiv

```
int execlp( const char *file, const char *arg,
                ...);
int execvp( const char *file, char *const argv[]);
```

  - PATH otsing, var. argumendid & PATH + argumentmassiiv

```
char* getenv(const char *name);
```

exec() perekond (näpunäited)

- Esimene argument peab olema käivitatava programmi **nimi**. "Päris" argumente sinna panna on vale.
- Argumendi ja keskkonnapointerite massiiv **peab** lõppema NULL pointeriga
- Kui te keskkonda ei edasta, antakse edasi jooksva programmi keskkond: st *mingi* keskkond on programmil igal juhul olemas
- **exec ei lõpeta.** Kontroll antakse programmile, mis sa argumendiks andsid ja sinu programmi enam ei täideta. Kui exec tagastab väärtuse, tekkis viga.

Näide (kataloogi sisu näitamine)

```
#include <stdio.h>
main()
{
int err;
if (err = execl("/bin/ls", "ls", "-l", "/etc",
NULL))
    printf("Err=%d\n",err);
else
    printf("Mind ei trükita kunagi välja\n");
}
```

fork()

- Kuna exec asendab olemasoleva protsessi, peame kuidagi saama ka teha uusi. Uue jaoks on käsk:

```
pid_t fork()      /* pid_t is an int */
```

- Loob lapsprotsessi uue PID-ga. Kutsume välja ühe korra, tagastume kaks korda.
- Uus protsess on väljakutsuva (*parent*) täpne koopia.
- Tagastab lapse puhul 0 ja vanema puhul PID
- Vanem saab -1 ja errno, kui ei õnnestu
- Laps saab vanema kasutaja tegeliku ja kehtiva ID, grupi tegeliku ja kehtiva ID, keskkonna, avatud failideskriptorid (kopeeritakse) jne...

wait()

- fork() ei oota, et laps oma töö lõpetaks

```
pid_t wait(int *status);
```

- status on pointer kohale, kuhu süsteem võiks salvestada lapse poolt väljumisel tagastatud väärtuse
- Tagastusväärtuseks on lõpnud lapse PID
- Ootab kuni laps on lõpetanud
- Kui laps lõpetab, muutub ta **zombiks**: kernel jätab osa tema kohta käivast infost meelde ja koristab ülejäänu ära. Kui vanem lõpetab lapse vastu huvi tundmata, teeb init protsess zombikoristuse ise ära.

Shelli näide

shell.c

exit()

```
void exit(int status);
int atexit(void (*func)(void));
```

- exit() lõpetab protsessi töö viisakalt ära. Vanemale tagastatakse väljumisväärtus, kõik atexit() funktsioonid kutsutakse välja registreerimisele vastupidises järjekorras ning kõik avatud stream'id *flush*itakse ning suletakse
- exit() ei tagastu
- atexit() registreerib argumendina antud funktsiooni exit() käsu puhul käivitamiseks
- Viisakas lõpp on kas exit() abil või return main() funktsioonist

atexit() näide

```
#include <stdio.h>

void final_wish(void);

main() {
     atexit(final_wish);
     printf("Hello World!\n");
}

void final_wish(void) {
   printf("Wish I could have lived forever\n");

}
```

Deemonid

- Deemonid (Daemon) on serveriprotsessid, mis jooksevad taustal. Näiteks võid oma FTP serveri deemonina programmeerida.

```
void run_server(void);
main()  {
int pid;
if ((pid = fork()) < 0) {
        perror("server process does not fork");
        exit(1);
   }
if (pid==0)
        run_server();
else {
       printf("FTP daemon successfully launched, pid=%d\n",
                        pid);
       exit(0);
      }
}
```

Protsessi omanik ja grupp

- *Real id* ütleb, kes sa tegelikult oled, *Effective id* ütleb, mis õigused on sul failidele ja seadmetele.

```
uid_t getuid(void);      /* uid_t on int */
uid_t geteuid(void);
```

- Tegelik id vastab väljakutsunud protsessi UIDle
- Kehtiv id vastab setuid biti poolt määratud UIDle

```
gid_t getgid(void);
gid_t getegid(void)
```

- Tegelik grupi id vastab väljakutsuva protsessi GIDle.
- Kehtiv grupi id vastab set ID biti poolt määratule

Protsessi identiteet

```
pid_t getpid(void);
pid_t getppid(void);
```

- getpid() tagastab parajasti jooksva protsessi PID (seda kasutatakse tihti nt ajutiste failide unikaalsete nimede loomisel)
- getppid() tagastab parajasti jooksva protsessi vanemprotsessi PID
- init on protsess nr 1.
- Kõik peale mõne üksiku kerneliprotssi põlvnevad initist

Valge slaid